CTGEN - a Unit Test Generator for C

Tatiana Mangels*
University of Bremen
Germany
tatiana@informatik.uni-bremen.de

Jan Peleska†
University of Bremen
Germany
jp@informatik.uni-bremen.de

We present a new unit test generator for C code, CTGEN. It generates test data for C1 structural coverage and functional coverage based on pre-/post-condition specifications or internal assertions. The generator supports automated stub generation, and data to be returned by the stub to the unit under test (UUT) may be specified by means of constraints. The typical application field for CTGEN is embedded systems testing; therefore the tool can cope with the typical aliasing problems present in low-level C, including pointer arithmetics, structures and unions. CTGEN creates complete test procedures which are ready to be compiled and run against the UUT. In this paper we describe the main features of CTGEN, their technical realisation, and we elaborate on its performance in comparison to a list of competing test generation tools. Since 2011, CTGEN is used in industrial scale test campaigns for embedded systems code in the automotive domain.

1 Introduction

Unit testing is a well-known method, widely used in practice, by which a single program function or method (hereafter referred to as *module* or the *unit under test (UUT)*) is tested separately with respect to its functional correctness[1]. The test data for performing a module test specifies initial values for input parameters, global variables, and for the data to be set and returned by sub-functions called by the UUT. The test results are typically checked by means of pre-/post-conditions, that is, logical conditions relating the program pre-state to its post state after the module's execution. More complex correctness conditions, such as the number of sub-function calls performed by the UUT may also be specified by means of pre-/post-conditions, if auxiliary variables are introduced, such as counters for the number of sub-function calls performed. The manual elaboration of test data and the development of test procedures exercising these data on the UUT is time consuming and expensive.

Symbolic execution [17, 10] is a well-known technique that addresses the problem of automatic generation of test inputs. Symbolic execution is similar to a normal execution process, the difference being that the values of program inputs are symbolic variables, not concrete values. When a variable is updated to a new value it is possible, that this new value is an expression over such symbolic variables. When the program flow comes to a branch, where the condition depends on a symbolic variable, this condition can be evaluated both to *true* or *false*, depending on the value of a symbolic variable. Through the symbolic execution of a path, it becomes a *path condition*, which is a conjunction of all branch conditions occurring on the path. To reason about path conditions and hence about feasibility of paths a constraint solver

*The author has been supported by Siemens AG through a research grant of the Graduate School on Embedded Systems GESy at the University of Bremen (http://www.informatik.uni-bremen.de/gesy)

†The author's research is funded by the EU FP7 COMPASS project under grant agreement no.287829

[1] In this paper we disregard tests investigating non-functional properties, such as worst case execution time or absence of run-time errors, since these are often more successfully investigated by means of formal verification, static analysis or abstract interpretation.

F. Cassez, R. Huuck, G. Klein and B. Schlich (eds.):
Systems Software Verification Conference 2012 (SSV 2012)
EPTCS 102, 2012, pp. 88–102, doi:10.4204/EPTCS.102.9

is used. When the solver determines a path condition as feasible, it calculates concrete values which can then be used as concrete inputs to explore the corresponding path.

In this paper we present CTGEN, an automatic test generation tool, based on symbolic execution. The objective of CTGEN is to cover every branch in the program, which is an undecidable problem, so in practice CTGEN tries to generate a test that produces as high a coverage for the module under tests as possible. For each unit under test CTGEN performs symbolic analysis and generates a test in RT-Tester syntax [13], which can be directly compiled and executed. For the definition of expected behaviour of the UUT CTGEN provides an annotation language, which also offers support for referencing functional requirements for the purpose of traceability from tests to requirements. Apart from atomic integral data types, CTGEN supports floating point variables, pointer arithmetics, structures and arrays and can cope with the typical aliasing problems in C, caused by array and pointer utilisation. Recursive functions, dynamic memory, complex dynamic data structures with pointers (lists, stacks etc.) and concurrent program threads are not supported. CTGEN does not check the module under test for run-time errors, but rather delegates this task to an abstract interpreter which is developed in our research group [21].

CTGEN does not rely on knowledge about all parts of the program (such as undefined or library functions). Where several other unit test automation tools [25, 8, 14] fall back to the invocation of the original sub-function code with concrete inputs if an external function occurs on the explored path, CTGEN automatically generates a *mock object* replacing the external function by a test stub with the same signature. Furthermore it calculates values for the stub's return data, output parameters and global variables which can be modified by the stubbed function in order to fulfil a path condition. In this way CTGEN can also simulate exceptional behaviour of external functions. It is possible but not required to customise stub behaviour by using pre- and post-conditions described in Section 3. If no restrictions were made, however, the stub's actions can deviate from the real behaviour of the external function.

Main contributions. CTGEN is distinguished from other unit test automation tools by the following capabilities: (1) handling of external function calls by means of mock objects, (2) symbolic pointer and offset handling, and (3) requirement tracing. As pointed out in [24], pointers and external function calls are the most important challenges for test data generation tools. CTGEN handles both as described in this paper. Furthermore we provide heuristics for the selection of paths through the UUT in such a way, that the coverage may be achieved with as few test cases as possible.

Overview. This paper is organised as follows. Section 2 presents an overview of the CTGEN architecture. Section 3 introduces an annotation language developed to define expected behaviour of the UUT and to support requirements tracing. In Section 4 we discuss test case generation strategies used by CTGEN. Section 5 describes the algorithm for handling pointer comparisons and pointer arithmetics. We show experimental results and compare CTGEN to other test data generation tools in Section 6, discuss related work in Section 7 and present a conclusion in Section 8.

2 Architecture

CTGEN is structured into two main components (see Fig. 1):

The *preprocessor* operates on the UUT code. It consists of (1) the CTGEN preprocessor transforming code annotations as described in Section 3, (2) a `GCC plugin` based on [19], compiling the prepared source code into a textual specification consisting of Control Flow Graphs (CFGs) in 3-address code

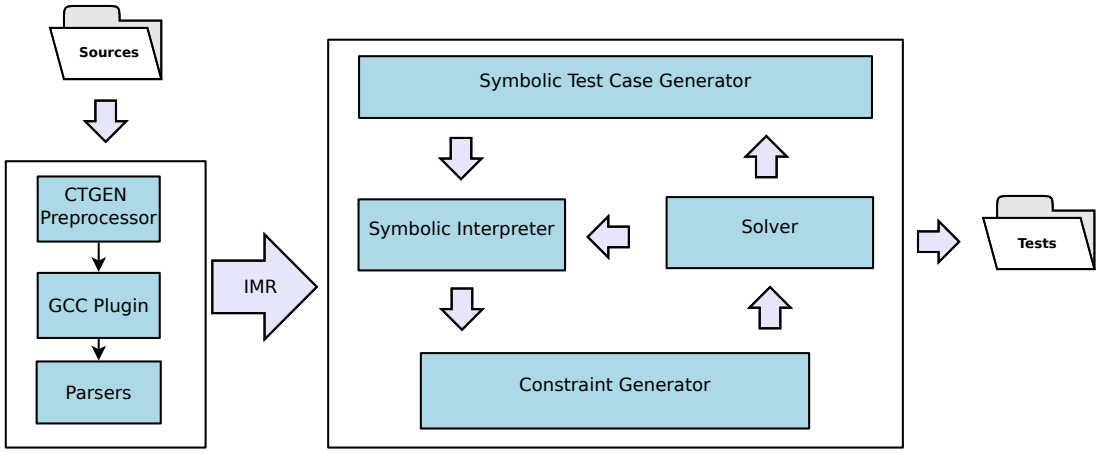

Figure 1: CTGEN overview.

and symbol table information like function signatures, types and variables, and (3) parsers, transforming CFGs and symbol table information into the Intermediate Model Representation (IMR).

The *analyser* operates on the IMR, its building blocks and their interactions are described below. The *Symbolic Test Case Generator* is responsible for lazy expansion of the CFGs related to the function under test and its sub-functions. Moreover, it handles the selection of paths, each beginning with the start node of the CFG, and containing yet uncovered transitions (for more details see Section 4). If such path can be found, it is passed to the *Symbolic Interpreter*, which traverses the path and symbolically calculates the effect of its statements in the memory model. As soon as the next node on the path is guarded by a non-trivial condition, the *Constraint Generator* [20] is called and resolves all pointers and array references occurring in this condition. It also passes the resulting constraint to the *Solver*. CTGEN uses a SMT solver, SONOLAR, which has been developed in our research group [21] and supports integral and floating point datatypes, arrays and bit vectors. If the solver can find a solution for the constraint, the solution is passed back to the Symbolic Interpreter, which continues to follow the path under investigation. Otherwise, if the constraint is infeasible or the path is completed, the solver passes the result to the *Symbolic Test Case Generator*. It learns from infeasibility and tries to produce another path containing still uncovered transitions, or tries to continue the completed path. When no such paths can be found, a unit test is generated based on the collected solutions (if any) and is stored in the file system.

3 Annotation Language

Annotations in CTGEN allow users to specify the expected behaviour of functions. For the definition of annotations we have chosen the approach used in sixgill [16]: they are specified as GCC macros which are understood by CTGEN. Thus the annotations can be turned on and off as needed. The arguments of the annotations follow standard C syntax, so that no additional expertise is required for the user. All annotations are optional – if there are none, CTGEN will try to cover all branches and detect unreachable code, using arbitrary type-compatible input data.

Pre- and post-conditions are defined as follows:

```
__rtt_precondition(PRE);
__rtt_postcondition(POST);
```

A precondition indicates, that the expected behaviour of the specified function is only guaranteed if the condition PRE is true. A post-condition specifies, that after the execution of a function the condition POST must hold. Pre- and post-conditions have to be defined directly at the beginning of a function body. PRE and POST are boolean C expressions, including function calls. All variables occurring in these conditions must be global, input or output parameters, or refer to the return value of the specified function. To specify conditions involving the return value of the UUT the CTGEN variable __rtt_return may be used. Annotation __rtt_initial(VARNAME) can be used within other annotation expressions – in particular, in post-conditions – for referring to the initial value of the variable VARNAME, valid before the function was executed.

To reason over local variables, auxiliary variables are used. Auxiliary variables can never occur in assignments to non-auxiliary variables or in control flow conditions [3, 20]. They can be defined as follows:

```
__rtt_aux(TYPE, VARNAME);
```

In this way an auxiliary variable of type TYPE with the name VARNAME will be declared and can be used in the following CTGEN annotations in the same way as regular variables.

For a more detailed specification of the expected behaviour of the function, test cases can be used:

```
__rtt_testcase(PRE, POST, REQ)
```

The argument PRE defines a pre-condition and the argument POST a post-condition of the current testcase. The argument REQ is a string tag defining a functional requirement that corresponds to the pre- and post-condition of this test case. If there is more than one requirement, they can be listed in a comma-separated list.

Users can specify global variables which are allowed for modification in the current function by means of annotation:

```
__rtt_modifies(VARNAME)
```

CTGEN traces violations, even in cases where a prohibited variable is modified by means of pointer dereferencing. For each breach of a modification rule an assertion is generated, which records the line number where the illegal modification occurred, e. g.

```
// violated var VARNAME in line(s) 1212, 1284
@rttAssert(FALSE);
```

The __rtt_assign(ASSIGNMENT) annotation is intended for assignments to auxiliary variables. In the following example an auxiliary variable a_aux is first declared using __rtt_aux(); it may then be used in a post-condition's expression. To define its value, __rtt_assign() is used in the function body.

```
__rtt_aux(int, a_aux);
__rtt_postcondition(a_aux == 0);
...
int b;
...
__rtt_assign(a_aux = b);
```

__rtt_assert(COND) can be used in different places of the function to ensure a specific property. If condition COND is seen to fail during test generation an assertion recording the line number where the violation occurs is inserted into the generated test.

An example of a specification of the expected behaviour of a function is illustrated in Fig. 2. The function `alloc()` returns a pointer `allocp` to `n` successive characters if there is still enough room in the buffer `allocbuf` and zero if this is not the case. First, by using `__rtt_modifies` we state that `alloc()` can only modify `allocp`, and modification of `allocbuf` is consequently prohibited. Annotation `__rtt_precondition` specifies that the expected behaviour of `alloc()` is guaranteed only if the parameter `n` is greater or equal zero and `allocp` is not a `NULL`-pointer. Furthermore the condition `__rtt_postcondition` states that after the execution of the function under test `allocp` must still be within the bounds of the array `allocbuf`. Finally, test cases are defined for the situations where (a) memory can still be allocated and (b) not enough memory is available.

```
1  char allocbuf[ALLOCSIZE];
2  char *allocp = allocbuf;
3
4  char *alloc(int n){
5    __rtt_modifies(allocp);
6    __rtt_precondition(n >= 0 && allocp != 0);
7    __rtt_postcondition(allocp != 0 && allocp <= allocbuf + ALLOCSIZE);
8    __rtt_testcase(allocbuf + ALLOCSIZE - __rtt_initial(allocp) < n,
9                   __rtt_return == 0,
10                  "CTGEN_001");
11   __rtt_testcase(allocbuf + ALLOCSIZE - __rtt_initial(allocp) >= n,
12                  __rtt_return == __rtt_initial(allocp),
13                  "CTGEN_002");
14
15   char *retval = 0;
16   if(allocbuf + ALLOCSIZE - allocp >= n){
17     allocp += n;
18     retval = allocp - n;
19   }
20
21   return retval;
22 }
```

Figure 2: Example: Specification of expected behaviour.

4 Symbolic Test Case Generation

As described in Section 2, the symbolic test case generator is responsible for the selection of test cases, which the symbolic interpreter then tries to cover. As a criteria to reason about completeness of testing we use code coverage. CTGEN supports the following two coverage criteria: (1) *statement coverage* (C0), which requires, that each statement in the program is executed at least once, and (2) *decision coverage* (C1), which requires that additionally to statement coverage each decision in the program was evaluated at least once to *true* and at least once to *false*.

A central data structure in the symbolic test case generator is a *Symbolic Test Case Tree* (STCT), which stores bounded paths through the control flow graph of the UUT. To build a STCT the CFG is expanded node by node. During expansion of CFG node n, each outgoing edge of n is analysed and each of its target nodes becomes a new corresponding STCT leaf, even if this target node already has been expanded. The nodes are labeled with a number k, so that (n,k) is a unique identifier of a STCT node, while n may occur several times in the STCT if it lies on a cyclic CFG path. The STCT root corresponds to the CFG start node (see [5] for further details).

When it is required to select a new test case, the symbolic test case generator takes an edge in the

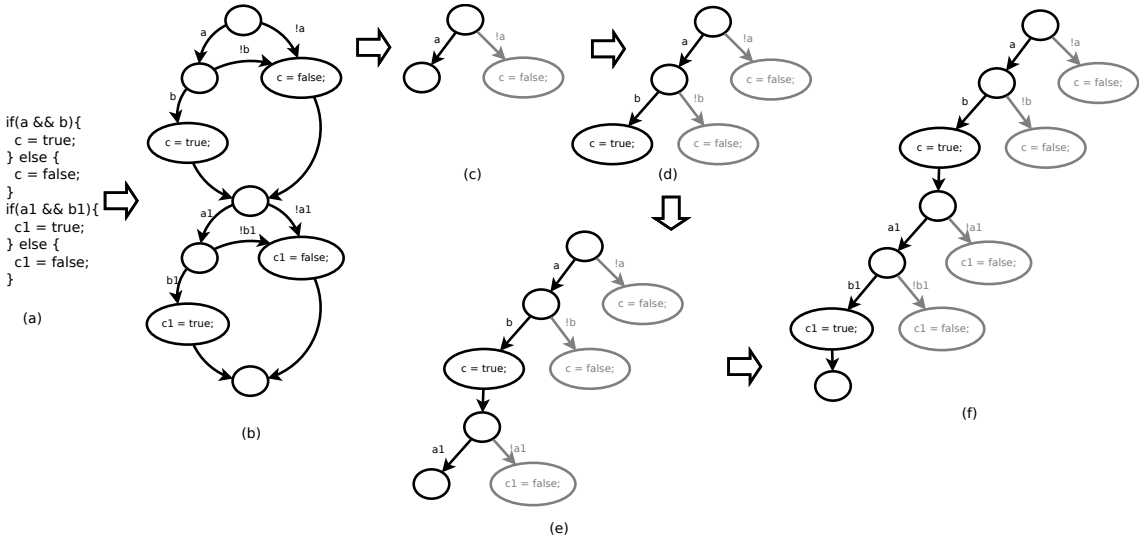

Figure 3: Expansion/Selection Example.

CFG, which is still uncovered. Subsequently it finds a corresponding STCT edge (since nodes n, n' may occur several times in the STCT, edges $n \to n'$ may occur several times as well) and traces it bottom-up to the start node. This trace is then returned to the test data generator by the symbolic test case generator for further investigation.

The depth-first search used by several test generating tools [6, 25, 14, 8] allows to reuse the information of the shared part of the execution trace, but on the other hand can cause the generator to get "stuck" analysing a small part of the program and generating a lot of new test cases but no (or little) new coverage. Pex [26] avoids using depth-first and backtracking techniques by storing the information of all previously executed paths. CTGEN behaves similarly: during the symbolic execution of a trace it stores information already gained in the form of computation histories and path constraints associated with each branch point of the trace under consideration.

Our expansion and selection strategies are motivated as follows: (a) The larger the explored range of the variable values participating in the guard conditions closest to the start node, the higher the probability to cover more branches depending on these variables further down the CFG. So we prioritise edges after their proximity to the start node. This allows achieving more coverage when it is not possible to explore the function completely due to its size. Furthermore this approach minimises the number of paths that must be explored to achieve 100 % C1 coverage, which in turn reduces the overall time for test generation. (b) A path is interesting for the test case generation for C1 coverage only if it contains still uncovered edges with non-trivial guard conditions, since otherwise no new coverage can be achieved by interpreting this path. So we expand until the STCT contains a new uncovered edge whose guard condition is not always *true*, or until no expansion can find any additional uncovered edges. At the same time we try to minimise the size of a STCT, so we stop the expansion process as soon as an uncovered edge with non-trivial guard condition occurs. We call this approach *incremental expansion*. (c) To further minimise the size of a STCT we incrementally expand only the end node of the path under consideration (initially the root node), select the continuation for it and hand it over to the solver. We continue this step by step until the path is complete. After that and according to prioritisation of edges, a new path

C code	path constraint	solver solution	generated test case

`void test(char *p1,`
`char *p2){`
`if(p1 < p2){`
`ERROR;`
`}`
`}`
	`p1@baseAddress == p2@baseAddress &&`
	`p1@offset < p2@offset &&`
	`0 <= p1@offset < 10 &&`
	`0 <= p2@offset < 10`
		`p1@baseAddress = 2147483648`
		`p2@baseAddress = 2147483648`
		`p1@offset = 0`
		`p2@offset = 7`
			`char* p1, p2;`
			`char p1__autogen_array[10];`
			`unsigned int p1__autogen_offset;`
			`unsigned int p2__autogen_offset;`
			
			`p1 = p1__autogen_array;`
			`p2 = p1__autogen_array;`
			
			`p1__autogen_offset = 0;`
			`p1 += p1__autogen_offset;`
			`p2__autogen_offset = 7;`
			`p2 += p2__autogen_offset;`
			
			`@rttCall(test(p1, p2));`

Table 1: Pointer handling example.

is selected. If the selected path is infeasible, the responsible branch is deleted from the STCT, and the selection/expansion process is continued with the alternative branch.

The loop constructs are unwound according to our expansion strategy: the loop body is incrementally expanded until the exit condition is reached. If the exit condition can be evaluated to *true*, the loop is exited. Otherwise it is further expanded. This process is bounded by a configurable command line parameter, that defines the maximal possible depth of the STCT.

A simple example of our expansion/selection strategy is illustrated in Fig. 3. Fig. 3(a) shows a program, its corresponding CFG is presented in Fig. 3(b). After the initial expansion (Fig. 3(c)) a trace consisting of edge $< a >$ is selected (nodes and edges, that belong to the trace are drawn black, those that do not belong are drawn grey). After the trace is interpreted and evaluated as feasible, its last node is expanded (Fig. 3(d)) and a new trace (the continuation of the last one) consisting of egdes $< a >$ and $< b >$ is selected. After this trace is interpreted as well, its last node $(c = true;)$ is expanded until the new edge whose guard condition is not always *true* appears (Fig. 3(e)). Afterwards a new trace (continuation of the previous one) is selected. This process continues until the path is complete (Fig. 3(f)). Finally a new trace, corresponding to the prioritisation of edges, will be selected (here $<!a>$) and the whole process will be repeated.

5 Symbolic Execution

Due to aliasing in C/C++ the value of a variable can be changed not only by a direct reference to its name, but also by assignments to de-referenced pointers pointing to this specific variable. In the case of arrays different index expressions may reference the same array element. This makes it difficult to identify variable changes along program paths involving pointer and array expressions. To solve this problem, a memory model consisting of a history of *memory items* was introduced in [20, 19]. Each memory item is defined by its base address, offset, length, value expression and time interval (measured in computation steps) where it is valid. Computations are defined as memory modifications which are stored in corresponding memory items. Furthermore, the stored values are not resolved to concrete or abstract valuations, but are specified symbolically. This approach allows not only to find the actual memory area where a new value is written to, but also enables us to handle pointer comparisons and pointer arithmetics as described below.

Pointer Handling We handle all memory areas pointed to by pointers as arrays with configurable size. By abstracting pointers to integers we achieve that constraints over pointers can be solved by a solver

C code

```
void test(int p1, int p2){
  __rtt_modifies(globalVar);
  globalVar = -p2;
  if(func_ext(p1) > p2 && func_ext(p2) == p1 &&
     globalVar == p2){
    ERROR;
  }
}
```

path constraint

```
func_ext@RETURN@0 > p2 &&
func_ext@RETURN@1 == p1 &&
globalVar@func_ext@1 == p2
```

solver solution

```
func_ext@RETURN@0 = 21
func_ext@RETURN@1 = 0
globalVar@func_ext@1 = -1
p1 = 0
p2 = -1
```

generated test case

```
extern unsigned int func_ext_STUB_testCaseNr;
extern unsigned int func_ext_STUB_retID;
extern int func_ext_STUB_retVal[2];

int p1, p2;

/***** STUB func_ext *****/
func_ext_STUB_testCaseNr = 0;
func_ext_STUB_retID = 0;

/* set values for return */
func_ext_STUB_retVal[0] = 21;
func_ext_STUB_retVal[1] = 0;
/***** end STUB func_ext *****/

p1 = 0;
p2 = -1;
@rttCall(test(p1, p2));
```

generated stub

```
int func_ext(int a){
  @GLOBAL:
    unsigned int func_ext_STUB_testCaseNr;
    unsigned int func_ext_STUB_retID;
    int func_ext_STUB_retVal[2];
  @BODY:
    func_ext_RETURN = func_ext_STUB_retVal[func_ext_STUB_retID%2];
    if(func_ext_STUB_testCaseNr == 0){
      if(func_ext_STUB_retID == 1){
        globalVar = -1;
      }
    }
    func_ext_STUB_retID++;
};
```

Table 2: External function handling example.

capable of integer arithmetics. Each pointer p is associated with a pair of unsigned integers (A,x), where A corresponds to a base address of the memory area p points into and x is its offset ($p = A + x$). Whenever two pointers $p_i = (A_i, x_i)$, $i = 1,2$ occur in an expression $(p_1\ \omega\ p_2)$ with some comparison operator ω, we construct the following constraint:

$$A_1 == A_2\ \&\&\ x_1\ \omega\ x_2\ \&\&\ 0 \le x_1 < dim(p_1)\ \&\&\ 0 \le x_2 < dim(p_2)\ \&\&\ dim(p_1) == dim(p_2).$$

where $A_1 == A_2$ ensures, that p_1 and p_2 point to members of the same memory portion, $x_1\ \omega\ x_2$ reflects the pointer expression and $0 \le x_i < dim(p_i)$ guarantees, that the pointer stays within array bounds. To illustrate how CTGEN handles pointer operations we use the simple example shown in Table 1. Consider function `test()`: to reach the line with an error, input pointers `p1` and `p2` should fulfil `p1 < p2`. The symbolic interpreter generates a path constraint according to our approach with the size for the auxiliary arrays configured by the user over the command line argument equal to 10 (if it is not configured, default size will be taken), because this memory size makes condition `p1 < p2` feasible (any size ≥ 2 would suffice). All auxiliary variables used here are of type `unsigned int` so that the solver can easily solve the generated path constraint. CTGEN considers the calculated base address as a unique identifier of an auxiliary array: so, if the identifier appears for the first time, a new array is created, if the identifier is already known, the corresponding auxiliary array is taken. In our example identifier 2147483648 appears for the first time in the solution for `p1@baseAddress`, so the new array `p1__autogen_array` is created and the pointer `p1` is initialised with it. When the same identifier appears in the solution for `p2@baseAddress`, it is already known to CTGEN, so `p2` is also initialised with `p1__autogen_array`. Then offset values are processed and pointers are modified accordingly. With this test input the erroneous code in `test()` is uncovered.

External Function Handling When an external function call appears on the path under consideration, the return value of this external function, its output parameters and all global variables allowed for modification are handled as symbolic *stub variables*. These symbolic stub variables can possibly be

	Executable Lines	Branches	Time	Nr of Test Cases	Lines Coverage	Branch Coverage
$f_1()$	714	492	31m27.098s	59	95,1 %	89,0 %
$f_2()$	50	30	0m1.444s	8	100 %	100 %
$f_3()$	11	4	0m0.228s	3	100 %	100 %

Table 3: Experimental results on some functions of HELLA software.

modified by this call. A stub variable holds the information about the stub function to which it belongs and if it corresponds to the return value, the output parameter or global variable, changed by this stub.

To illustrate how this algorithm works we use the example shown in Table 2. In function `test()` the external function `func_ext()` is called twice. To reach the line with an error, `func_ext()` must return a value that is greater than the value of the parameter `p2` by the first call. Furthermore, by the second call it must return a value that is equal to the value of the parameter `p1`. The symbolic interpreter analyses, what could possibly be altered by `func_ext()` and creates the stub variables `func_ext@RETURN` and `globalVar@func_ext`. The constraint generator generates a path constraint as shown in Table 2. The occurences of the stub variables are versioned (actually, in CTGEN all variables in path constraint are versioned [20], but this is ignored here to keep the example simple). The version of a stub variable corresponds to the running number of the call of the external function. Here `func_ext@RETURN@0` corresponds to the return value of the first call and `func_ext@RETURN@1` to the return value of the second one. Now consider the generated test case and the generated stub in Table 2. As was already mentioned, CTGEN generates tests in RT-Tester syntax. An array `func_ext_STUB_retVal` of size two (corresponding to the number of calls of the function `func_ext()`) is created to hold the calculated return values. These values are stored by the test driver according to their version. The variable `func_ext_STUB_retID` corresponds to the running number of the stub call. It is reset by the test driver before each call of UUT and incremented by the corresponding stub each time it is called. Since one test driver can hold many test cases, the variable `func_ext_STUB_testCaseNr` that corresponds to the number of test case is created. This variable is set by the test driver. The value of the global variable `globalVar` is set by the stub if the number of the stub call and the test case number match the calculated ones for this global variable.

A detailed description how to perform symbolic execution in presence of arrays and primitive datatypes can be found in [20].

6 Experimental Results

The experimental evaluation of CTGEN and comparison with competing tools was performed both with synthetic examples evaluating the tools' specific pointer handling capabilities and with embedded systems code from an industrial automotive application. The latter presented specific challenges: (1) the code was automatically generated from Simulink models. This made automated testing mandatory since small model changes considerably affected the structure of the generated code, so that re-use of existing unit tests was impossible if models had been changed. (2) Some units were exceptionally long because insufficient HW resources required to reduce the amount of function calls.

Table 3 shows the results achieved by CTGEN in the automotive test project on some selected functions. The most challenging function was f_1 with over 2000 lines of code (714 executable lines), using structures, bit vectors, pointer parameters and complex branch conditions. Nevertheless CTGEN was able to generate 95,1% line and 89,0% branch coverage with 59 automatically generated test cases. Furthermore, by using preconditions as guides for CTGEN to cover parts of code further down in the CFG, it was possible to increase the coverage even more. Function f_2 with 50 executable lines of code (about 300

lines of code) represents a typical function in the project. For such functions CTGEN achieved 100% C1 coverage. Function f_3 includes pointer comparison, pointer dereferencing and a `for`-loop with an input parameter as a limit. However, due to small branching factor CTGEN achieves 100% coverage with only 3 test cases and a generation time under one second. Summarising, CTGEN proved to be efficient for industrial test campaigns in the embedded domain and considerably reduced the over all project efforts.

In comparison (see Table 4), experiments with KLEE [7] and PathCrawler [6] demonstrated that CTGEN delivers competitive results and outperformed the others for the most complex function $f_1()$. The experiments with PathCrawler were made with the online version [1], so it was not possible to exactly measure the time spent by this tool. This tool, however, could not handle the complexity of $f_1()$, and KLEE did not achieve as much coverage as CTGEN, we assume that this is due to the path-coverage oriented search strategy which has not been optimised for achieving C1 coverage.

Functions $f_4()$ and $f_5()$ are also taken from the automotive testing project. Function $f_5()$ has struct-inputs with bit fields. KLEE achieved 100 % path coverage. PathCrawler also targets path coverage, but due to limitations of the online version (number of generated test cases, available amount of memory) delivers only 201 test cases. However, we assume that without these limitations it will also achieve 100 % path coverage, although in a larger amount of time than KLEE. For the example function $Tritype()$ KLEE delivers bad results because it does not support floating types; there is, however, an extension KLEE-FP [11] targeting this problem. PathCrawler excels CTGEN and KLEE, but can handle only the `double` type, not `float`, while CTGEN can calculate bit-precise solutions for both. $alloc\_ptr()$ and $comp\_ptr()$ demonstrate handling of symbolic pointers which is not supported by PathCrawler; KLEE and CTGEN deliver comparable results.

7 Related Work

The idea of using symbolic execution for test data generation is not new, it is an active area of research since the 70's [17, 10]. In the past a number of test data generation tools [7, 8, 6, 4, 18, 2, 26, 25, 14, 15, 23] were introduced. Nevertheless to the best of our knowledge, only Pex (with Moles) supports automatic stub generation as provided by CTGEN. Furthermore, CTGEN seems to be the only tool supporting traceability between test cases and requirements. From the experimental results available from other tool evaluations we conclude that CTGEN outperforms most of them with respect to the UUT size that still can be handled for C1 coverage generation.

DART [14] is one of the first concolic testing tools to generate test data for C programs. It falls back to concrete values by external function calls, and does not support symbolic pointers. CUTE [25] is also a concolic test data generator for C, and, like DART, falls back to concrete values by external function calls. It supports pointers, but collects only equalities/inequalities over them, while CTGEN supports all regular pointer arithmetic operations.

SAGE [15] is built on DART, is a very powerful concolic testing tool utilising whitebox fuzzing. It is fully automated and is in daily use by Windows in software development process, and, accordingly to authors, uncovered about half of all bugs found in Windows 7. SAGE has a precise memory model, that allows accurate pointer reasoning [12] and is very effective because it works on large applications and not small units, which allows to detect problems across components. Nevertheless SAGE uses concrete values for sub-function calls which cannot be symbolically represented and, to our best knowledge, does not support pre- and post-conditions.

Pex [26] is an automatic white-box test generation tool for .NET, developed at Microsoft Research. It generates high coverage test suites applying dynamic symbolic execution for parameterised unit tests

	CTGEN	KLEE		PathCrawler
$f_1()$ (714 lines, 492 branches)				
Time	31m27.098s	16m50s	586m16.590s	-
Nr of Test Cases	59	1120	24311	-
Lines Coverage	95,1 %	77,9 %	78,54 %	-
Branch Coverage	89,0 %	58,2 %	59,36 %	-
$f_4()$ (19 lines, 4 branches)				
Time	0.062s	0.040s		$<$ 1s
Nr of Test Cases	3	3		9
Lines Coverage	100 %	100 %		100 %
Branch Coverage	100 %	100 %		100 %
$f_5()$ (28 lines, 35 branches)				
Time	0.337s	3.234s	41,176s	10s
Nr of Test Cases	3	463	2187	201
Lines Coverage	100 %	100 %	100 %	85,71 %
Branch Coverage	100 %	100 %	100 %	85,71 %

	CTGEN	KLEE	PathCrawler
Tritype()			
Time	8.404	0.095	$<$ 1s
Nr of Test Cases	8	1	11
Lines Coverage	100 %	41,66 %	100 %
Branch Coverage	100 %	20 %	100 %

```
int Tritype(double i, double j, double k){
  int trityp = 0;
  if (i < 0.0 || j < 0.0 || k < 0.0)
    return 3;
  if (i + j <= k || j + k <= i || k + i <= j)
    return 3;
  if (i == j) trityp = trityp + 1;
  if (i == k) trityp = trityp + 1;
  if (j == k) trityp = trityp + 1;
  if (trityp >= 2)
      trityp = 2;
  return trityp;
}
```

	CTGEN	KLEE	PathCrawler
alloc_ptr()			
Time	0.071s	0.064s	$<$ 1s
Nr of Test Cases	4	4	2
Lines Coverage	100 %	100 %	42,86 %
Branch Coverage	100 %	100 %	50 %

```
char *alloc_ptr(char *allocbufp, char *allocp,
                unsigned int n)
{
    if(allocbufp == 0 || allocp == 0)
        return 0;

    if(allocbufp + ALLOCSIZE - allocp >= n){
        allocp += n;
        return allocp - n;
    }
    return 0;
}
```

	CTGEN	KLEE	PathCrawler
comp_ptr()			
Time	0.032s	0.055s	$<$ 1s
Nr of Test Cases	4	4	2
Lines Coverage	100 %	100 %	75 %
Branch Coverage	100 %	100 %	50 %

```
int comp_ptr(char *p1, char *p2)
{
    if(p1 != NULL && p2 != NULL && p1 == p2){
        return 1;
    }
    return 0;
}
```

Table 4: Experimental results compared with other tools.

(PUT). Similarly to CTGEN it uses annotations to define expected results, and the Z3 SMT Solver to decide on feasibility of execution paths. It also supports complex pointer structures [27]. Pex with Moles also offers a possibility to handle external function calls with stubs; however, it is required, that the mock object is first generated by the user with Moles to enable Pex to handle such calls, while CTGEN does it automatically.

Another approach using symbolic execution is applied by KLEE [7], the successor of EXE [8]. KLEE focuses on the interactions of the UUT with the running environment – command-line arguments, files, environment variables etc. It redirects calls accessing the environment to *models* describing external functions in sufficient depth to allow generation of the path constraints required. Therefore KLEE can handle library functions symbolically only if a corresponding model exists, and all unmodelled library and external function calls are executed with concrete values. This may reduce the coverage to be generated, due to random testing limitations. Furthermore KLEE does not provide fully automated detection of inputs: they must be determined by the user either by code instrumentation or by the command line argument defining the number, size and types of symbolic inputs.

Pathcrawler [6] is another concolic testing tool. It tries to generate path coverage for C functions. In contrast to CTGEN, it only supports one dimensional arrays and does not support pointer comparisons and external function calls. Pathcrawler proposed an approach, similar to ours for handling of external function calls. Though, like Pex, it requires mock objects to be defined first by the user, together with a formal specification of its intended behaviour.

Another approach to test data generation in productive industrial environments is based on bounded model checking [2]. The authors used CBMC [9], a Bounded Model Checker for ANSI-C and C++ programs, for the generation of test vectors. The tool supports pointer dereferencing, arithmetic, dynamic memory and more. However, since CBMC is applied here to generate a test case for each block of the CFG of the UUT, CTGEN will be able to achieve full decision coverage with fewer test cases in most situations. For handling external function calls the authors of [2] use nondeterministic choice functions available in CBCM as stubs, and CBCM evaluates all traces resulting from all possible choices. However, the tool can only simulate return values of external functions and does not consider the possibility of manipulating values of global variables. Although CBMC allows assertions and assumptions in the function body, the authors use them only to achieve the branch coverage, not for checking functional properties.

PathFinder [23] is a symbolic execution framework, that uses a model checker to generate and explore different execution paths. PathFinder works on Java byte code, one of its main applications is the production of test data for achieving high code coverage. PathFinder does not address pointer problems since these do not exist in Java. For handling of external function calls the authors propose *mixed concrete-symbolic solving* [22], which is more precise than CTGEN's solution with stubs - it will not generate test data that is impossible in practise, but is incomplete, i.e. there exist feasible paths, for which mixed concrete-symbolic solving fails to find a solution. Furthermore, by definition of accurate pre- and post-conditions the problem with impossible inputs can be avoided by CTGEN.

Table 5 summarises the results of our comparison.

8 Conclusion

In this paper we presented CTGEN, a unit test generator for C functions. CTGEN aims at test data generation for C0 or C1 coverage and supports test specifications with pre- and post-conditions and other annotation techniques, automated stub generation, as well as structure, pointer and array handling in the

	CTGEN	PEX	CUTE	KLEE	PathCrawler	CBMC for SCS	DART	SAGE	PathFinder
Platform	Linux	Windows	Linux	Linux	Linux		Linux	Windows	Linux
Language	C	.NET	C	C	C	C	C	machine code	Java
CAPABILITIES									
C0	Y	Y	Y	Y	Y	Y	Y	Y	Y
C1	Y	Y	Y	Y	Y	Y	Y	Y	Y
MC/DC	N	Y	N	N	N	N	N	N	Y
C2	N		Y	Y	Y	N	Y	Y	Y
Pre-/Post	Y	Y	Y	Y	Y	N	N	N	Y
Requirements tracing	Y	N	N	N	N	N	N	N	N
Auxiliary vars	Y	N	NA	N	N	N	N	N	N
Pointer arithmetics	Y	Y	N	Y	Y	Y	N	Y	-
Pointer dereferencing	Y	Y	N	Y	Y	Y	N	Y	-
Pointer comparison	Y	Y	Y	Y	N	Y	N	Y	-
Function pointer	N	NA	N	NA	N	Y	N	NA	
Arrays	Y	Y	Y	Y	P	Y		Y	Y
Symbolic offset	Y	NA	N	Y			N	Y	
Complex dynamic data structures (lists...)	N	Y	Y		N	Y	N		Y
External function calls	Y	P	N	P	N	Y	N	N	P
Automatic stub handling	Y	P	N	N	N	N	N	N	N
Float/double	Y	N	N	N	Y	N	N	Y	Y
Recursion	N	Y	NA	Y	N	Y			Y
Multithreading	N	N	Y	N	N	N	N		Y
Automatic detection of inputs	Y	Y	N	N	Y	Y	Y	Y	
TECHNIQUES									
SMT solver	SONOLAR	Z3	lp_solver	STP	COLIBRI		lp_solver	Z3	choco, IASolver, CVC3
Concolic testing	N	Y	Y	Y	Y	N	Y	Y	N
STCT or acyclic graph with reNuse of nodes	STCT	STCT		application states		transition relation			Y
Depth-first search	N	N	Y	N	Y	N	Y	N	N

Table 5: Test Data Generating Tools.

UUT. The tool is targeted at embedded systems software testing. Its fitness for industrial scale testing campaigns has been demonstrated in a project from the automotive domain, where it effectively reduced time and effort spent during the testing phase. Further experiments with other test data generation tools showed that CTGEN produced competitive results and in some aspects outperformed the others.

To further improve performance and scalability of the tool we plan to replace the current test case tree structure used for investigation of function paths to be covered by acyclic graphs allowing the re-use of nodes. The test generation technique is currently extended to support MC/DC coverage.

References

[1] *PathCrawler*. Available at `http://pathcrawler-online.com/`. Last visited September 2012.

[2] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu & Salvatore Sabina (2010): *Using Bounded Model Checking for Coverage Analysis of Safety-Critical Software in an Industrial Setting*. Journal of Automated Reasoning 45, pp. 397–414, doi:10.1007/s10817-010-9172-3.

[3] Krzysztof R. Apt & Ernst-Rüdiger Olderog (1991): *Verification of Sequential and Concurrent Programs*. Springer, doi:10.1007/978-1-84882-745-5.

[4] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant & Dawn Song (2011): *Statically-directed dynamic automated test generation*. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, ACM, New York, NY, USA, pp. 12–22, doi:10.1145/2001420.2001423.

[5] Bahareh Badban, Martin Fränzle, Jan Peleska & Tino Teige (2006): *Test automation for hybrid systems*. In: *Proceedings of the 3rd international workshop on Software quality assurance*, SOQUA '06, ACM, New York, NY, USA, pp. 14–21, doi:10.1145/1188895.1188902.

[6] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger & N. Williams (2009): *Automating structural testing of C programs: Experience with PathCrawler*. In: *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, pp. 70 –78, doi:10.1109/IWAST.2009.5069043.

[7] Cristian Cadar, Daniel Dunbar & Dawson Engler (2008): *KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs*. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, USENIX Association, Berkeley, CA, USA, pp. 209–224. Available at `http://dl.acm.org/citation.cfm?id=1855741.1855756`.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill & Dawson R. Engler (2008): *EXE: Automatically Generating Inputs of Death*. *ACM Trans. Inf. Syst. Secur.* 12(2), pp. 10:1–10:38, doi:10.1145/1455518.1455522.

[9] Edmund Clarke, Daniel Kroening & Flavio Lerda (2004): *A Tool for Checking ANSI-C Programs*. In Kurt Jensen & Andreas Podelski, editors: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, *Lecture Notes in Computer Science* 2988, Springer, pp. 168–176, doi:10.1007/978-3-540-24730-2_15.

[10] L.A. Clarke (1976): *A System to Generate Test Data and Symbolically Execute Programs*. *Software Engineering, IEEE Transactions on* SE-2(3), pp. 215 – 222, doi:10.1109/TSE.1976.233817.

[11] Peter Collingbourne, Cristian Cadar & Paul H.J. Kelly (2011): *Symbolic crosschecking of floating-point and SIMD code*. In: *Proceedings of the sixth conference on Computer systems*, EuroSys '11, ACM, New York, NY, USA, pp. 315–328, doi:10.1145/1966445.1966475.

[12] Bassem Elkarablieh, Patrice Godefroid & Michael Y. Levin (2009): *Precise pointer reasoning for dynamic test generation*. In: *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, ACM, New York, NY, USA, pp. 129–140, doi:10.1145/1572272.1572288.

[13] Verified Systems International GmbH (2012): *RT-Tester 6.x: User Manual*.

[14] Patrice Godefroid, Nils Klarlund & Koushik Sen (2005): *DART: directed automated random testing*. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, ACM, New York, NY, USA, pp. 213–223, doi:10.1145/1065010.1065036.

[15] Patrice Godefroid, Michael Y. Levin & David A. Molnar (2008): *Automated Whitebox Fuzz Testing*. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, The Internet Society.

[16] Brian Hackett (2010): *sixgill*. Available at `http://sixgill.org/`. Last visited August 2012.

[17] James C. King (1976): *Symbolic execution and program testing*. *Commun. ACM* 19(7), pp. 385–394, doi:10.1145/360248.360252.

[18] Guodong Li, Indradeep Ghosh & Sreeranga Rajan (2011): *KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification*, *Lecture Notes in Computer Science* 6806, Springer Berlin / Heidelberg, pp. 609–615, doi:10.1007/978-3-642-22110-1_49.

[19] Helge Löding & Jan Peleska (2008): *Symbolic and Abstract Interpretation for C/C++ Programs*. *Electronic Notes in Theoretical Computer Science* 217(0), pp. 113 – 131, doi:10.1016/j.entcs.2008.06.045. Available at `http://www.sciencedirect.com/science/article/pii/S1571066108003885`.

[20] Jan Peleska (2010): *Integrated and Automated Abstract Interpretation, Verification and Testing of C/C++ Modules*. In Dennis Dams, Ulrich Hannemann & Martin Steffen, editors: *Concurrency, Compositional-*

ity, and Correctness, *Lecture Notes in Computer Science* 5930, Springer Berlin / Heidelberg, pp. 277–299, doi:10.1007/978-3-642-11512-7_18.

[21] Jan Peleska, Elena Vorobev & Florian Lapschies (2011): *Automated test case generation with SMT-solving and abstract interpretation*. In: *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, Springer-Verlag, Berlin, Heidelberg, pp. 298–312, doi:10.1007/978-3-642-20398-5_22. Available at `http://dl.acm.org/citation.cfm?id=1986308.1986333`.

[22] Corina S. Păsăreanu, Neha Rungta & Willem Visser (2011): *Symbolic execution with mixed concrete-symbolic solving*. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, ACM, New York, NY, USA, pp. 34–44, doi:10.1145/2001420.2001425.

[23] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person & Mark Pape (2008): *Combining unit-level symbolic execution and system-level concrete execution for testing nasa software*. In: *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, ACM, New York, NY, USA, pp. 15–26, doi:10.1145/1390630.1390635.

[24] Xiao Qu & Brian Robinson (2011): *A Case Study of Concolic Testing Tools and their Limitations*. In: *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, IEEE Computer Society, Washington, DC, USA, pp. 117–126, doi:10.1109/ESEM.2011.20.

[25] Koushik Sen, Darko Marinov & Gul Agha (2005): *CUTE: a concolic unit testing engine for C*. In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, ACM, New York, NY, USA, pp. 263–272, doi:10.1145/1081706.1081750.

[26] Nikolai Tillmann & Jonathan de Halleux (2008): *Pex–White Box Test Generation for .NET*. In Bernhard Beckert & Reiner Hähnle, editors: *Tests and Proofs*, *Lecture Notes in Computer Science* 4966, Springer Berlin / Heidelberg, pp. 134–153, doi:10.1007/978-3-540-79124-9_10.

[27] Dries Vanoverberghe, Nikolai Tillmann & Frank Piessens (2009): *Test Input Generation for Programs with Pointers*. In Stefan Kowalewski & Anna Philippou, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 5505, Springer Berlin / Heidelberg, pp. 277–291, doi:10.1007/978-3-642-00768-2_25.